Core Java Data Objects

By Sameer Tyagi, Michael Vorburger, Keiron McCammon, Heiko Bobzin

Published Sep 11, 2003 by Prentice Hall. Part of the Core Series series.

More Info

List Price: $49.99

Your Price: **$44.99** Ships for free! Save more by becoming a member.

Add To My Wish List

Already own this? Receive a coupon or access to bonus material when you register it.

Description | Extras | Reviews | Sample Content

Online Sample Chapter

Getting Started with JDO

Sample Chapter(s)

Download the Sample Chapter related to this title.

Table of Contents

Other

Chapter 3

Getting Started with JDO

"The expert at anything was once a beginner."
- Hayes

Using JDO to build an application that creates, reads, updates, and deletes persistent instances of Java classes is easy and requires only some basic knowledge about how JDO works and how to use it. Armed with this knowledge, you can develop your first JDO application and persist instances of Java classes transparently in a datastore. This chapter is a guide to getting started with using JDO, providing an understanding of how JDO works and how to use the basic APIs, and exploring some of the more advanced concepts related to using JDO.

This chapter covers the following topics:

- How JDO is able to transparently persist instances of Java classes
- The basic JDO interfaces and how they are related
- How to define a Java class that can be used with a JDO implementation
- How to connect to a datastore

- How to create, read, update, and delete persistent objects
- The types of fields, system classes, collection classes, and inheritance supported by JDO
- How to handle exceptions within an application
- The concept of object identity
- The different types of identity that can be used
- How concurrency control is enforced between multiple applications

The examples for this chapter can be downloaded from the Internet at www.corejdo.com and are located in the `com.corejdo.examples.chapter3` package. In many cases, the code snippets shown are simplified versions of the actual classes to allow the examples to focus only on the relevant concepts.

3.1 How Does JDO Work?

The goal of the JDO is to allow a Java application to transparently store instances of any user-defined Java class in a datastore and retrieve them again, with as few limitations as possible. This book refers to the instances that JDO stores and retrieves as *persistent objects*. From the application perspective, these persistent objects appear as regular, in-memory Java objects. However, the fields of these instances are actually stored in some underlying datastore persistently—all without any explicit action on behalf of the application.

JDO has nothing to do with where methods are executed; it does not provide a means of remote method invocation à la RMI and EJB, nor does it store and execute methods in some datastore. JDO simply specifies how the fields of a persistent object should be managed in-memory, being transparently stored to and retrieved from an underlying datastore. With JDO, methods are invoked on a persistent object by an application, as per any regular in-memory Java object.

Figure 3-1 provides a schematic of how JDO works.

The JDO implementation and the application run together in the same JVM. The application delegates to the JDO implementation to retrieve the fields of persistent objects as needed. The JDO implementation tracks modifications to the fields and writes these changes back to the datastore at the end of the transaction. The JDO implementation is responsible for mapping the fields of the persistent objects to and from memory and the underlying datastore.

JDO achieves transparency of access by defining a contract to which a class must adhere. Any class that implements this contract can then be used with any JDO implementation. JDO requires that a JDO implementation ensure that any class

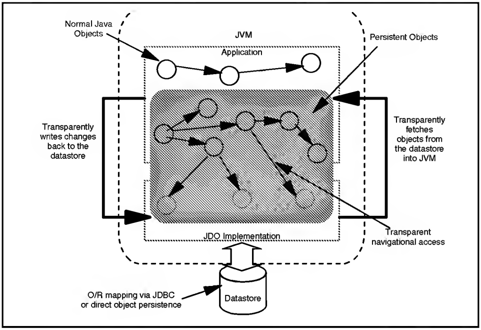

Figure 3–1 JDO runtime environment.

that adheres to the JDO persistence-capable contract can be used with any JDO implementation, without recompilation.

The ability to run a JDO application with any JDO implementation is akin to using JDBC, because a JDBC application can be run "as is" using JDBC drivers from different vendors and even using different relational databases. In fact, it's somewhat better than this, because with JDBC an application is still prone to differences in SQL support across different databases. With JDO, SQL is not directly exposed. Although a JDO runtime may itself use JDBC to access a relational database as its datastore, it is the responsibility of the JDO implementation to resolve the differences in SQL support across databases.

Even better, unlike SQL, a JDO application can work "as is" across different types of databases, not just relational: object databases, flat-files, and so on. All that is required is a JDO implementation.

The JDO specification defines the persistence-capable contract as a Java interface, called `PersistenceCapable`, and a programming style that the class implementation must follow. A class that adheres to this contract is referred to as being "persistence-capable."

A class is said to be persistence-capable if its instances can be stored in a datastore by a JDO implementation. However, just because a class is persistence-capable doesn't mean that all its instances have to be persistent; it just means the option is there. Whether a particular instance is persistent depends on the application. It's similar to Java serialization: Just because a class implements the `Serializable` interface doesn't mean that all its instances have to be serialized.

However, the intention of JDO is not to expect the developer to have to worry about making a class persistence-capable; it's a tedious job better left to tooling.

You can create a persistence-capable class in three main ways:

Source code generation: With this method, the source code for a class is generated from scratch. This approach works well if the object model is defined in a modeling tool and is being automatically generated, or the datastore schema already exists and the object model can be generated from it. Tools supplied by the JDO implementation would be used to generate source code adhering to the persistence-capable contract. The drawback of this approach is that it won't work for existing classes and won't appeal to those who like to write their own code.

Source code preprocessing: With this method, existing source code is preprocessed and updated. This approach works well if the source code for a class is available. Tools supplied by the JDO implementation would be used to read the original source code and update it to adhere to the persistence-capable contract. The drawback of this approach is that it won't work unless the original source code is available, but it does have the benefit that a developer can write his or her own source code. Typically, the preprocessing is a precompilation step in the build process, and the generated code may be kept to aid in debugging.

Byte code enhancement: With this method, the compiled Java byte code for a class is enhanced directly. This approach works well even if the source code is not available. Tools supplied by the JDO implementation would be used to read a class file and insert additional byte code directly to make the class adhere to the persistence-capable contract. This approach has the benefit of being completely transparent to the developer, and the enhancement is simply a post-compilation step in the build process. Although the JDO specification requires that an enhanced class still function correctly when debugged against the original source code, some developers may be distrustful if they can't see the actual code for what has been changed (although they could, of course, always decompile the enhanced class file afterward).

Byte code enhancement is the approach used by the JDO reference implementation available from SUN Microsystems, and the enhancement tool is available for any developer to use. Some JDO implementations may provide their own enhancement tools also. Figure 3-2 provides a schematic of how the byte code enhancement process works:

The Java classes are compiled using a Java compiler to generate class files. The byte code enhancement tool reads the class files along with the JDO metadata for the classes (this metadata is explained in Section 3.3.1) and either updates the existing class files or creates new ones. The "enhanced" class files are then loaded by a JVM along with the JDO implementation and the application. The application can then use JDO to store instances of the persistence-capable classes in the datastore.

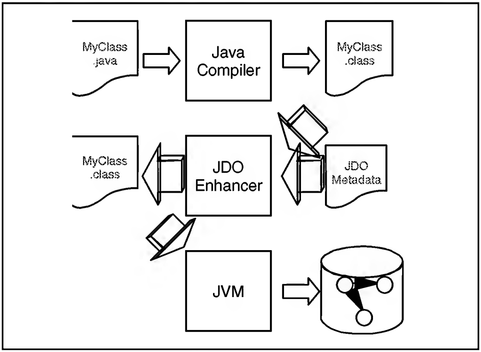

Figure 3–2 The byte code enhancement process.

3.2 The JDO Basics

The JDO specification defines 20 or so classes and interfaces in total, but a developer needs to know only five main classes (which are actually Java interfaces):

- PersistenceManagerFactory
- PersistenceManager
- Extent
- Query
- Transaction

Figure 3-3 provides a simplified class diagram showing how these interfaces are related:

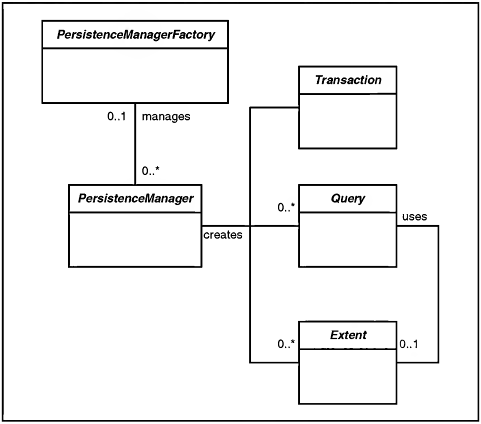

Figure 3–3 A simplified JDO class diagram.

A **PersistenceManagerFactory** is used to get a PersistenceManager instance. There is a one-to-many relationship between the PersistenceManagerFactory and the PersistenceManager. A PersistenceManagerFactory can create and manage many PersistenceManager instances and even implement pooling of Persistence-Manager instances.

A **PersistenceManager** embodies a connection to a datastore and a cache of in-memory persistent objects. The PersistenceManager interface is the primary means by which the application interacts with the underlying datastore and in-memory persistent objects.

From a PersistenceManager, the application can get one or more **Query** instances. A Query is how the application can find a persistent object by its field values.

In addition to a Query, the application can get an **Extent** from a PersistenceManager. An Extent represents all the instances of a specified class (and optionally subclasses) stored in the datastore and can be used as input to a Query or iterated through in its own right.

A **Transaction** allows the application to control the transaction boundaries in the underlying datastore. There is a one-to-one correlation between a Transaction and a PersistenceManager; there can be only one ongoing transaction per Persistence-Manager instance. Transactions must be explicitly started, and either committed or aborted.

3.3 Defining a Class

The first step in using JDO is to create a persistence-capable Java class. For the purposes of the examples in this book, it is assumed that byte code enhancement is being used, so all that is needed is a Java class. The following code snippet shows the code for a class called Author, which represents the author of a book:

```
package com.corejdo.examples.model;

public class Author {

  private String name;

  public Author (String name) {

      this.name = name;
  }

  protected Author () {}

  public String getName () {
```

```
    return name;
  }

  public void setName (String name) {

    this.name = name;
  }
}
```

This class is relatively simply with just one string field. JDO can, of course, support much more than this; see Section 3.9 for more details on what is supported.

Note that the fields of the class are declared as private. Unlike other approaches to object persistence, JDO does not require fields of classes to be declared public. In fact, JDO places very few constraints on the definition of the Java classes themselves.

There is one requirement, however: JDO requires that a persistence-capable class have a no-arguments constructor. This constructor does not need to be declared public; it just needs to be accessible to the class itself and to any potential subclasses. If there are no subclasses, it can be declared as private. However, if there are subclasses, it should be declared as protected or packaged so that it is accessible from the subclasses.

The no-args constructor is used by the JDO runtime to create empty instances of the class prior to retrieving its fields from the datastore.

To make the Author class persistence-capable, it must be compiled and passed to the byte code enhancement tool. Before this can be done, however, an XML file must be created that contains the JDO metadata for the class.

3.3.1 JDO metadata

For each persistence-capable class, JDO requires additional metadata. This metadata is specified in an XML file. The metadata is primarily used when making a Java class persistence-capable (by byte code, source enhancement, or code generation). However, it is likely also to be used by a JDO implementation at runtime.

By convention, these files have a `.jdo` suffix. Each class can have a metadata file, in which case the file is named after the class itself, or there can be one metadata file per package, in which case the file is named `package.jdo`. The metadata files should be accessible at runtime as resources via the class loader that loaded the persistence-capable Java class.

For the Author class, a file called `Author.jdo` would be located in the same directory as the Author class file. This allows the byte code enhancement process and JDO implementation to easily locate it:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"/>
  </package>
</jdo>
```

Alternately, the metadata for this class could be located in a file called `package.jdo` that would be located in the `com/corejdo/examples/model` directory. This file could contain metadata on all the classes in the "model" package (and sub-packages), not just the Author class. Appendix B provides a detailed overview of the syntax for the JDO metadata. For this class, however, all that is needed is to specify the package name and class name.

When the Author source and metadata files are ready, they can be passed to the byte code enhancement tool. This takes care of implementing the persistence-capable contract. As an example, if using the JDO reference implementation (RI), the Author class would be compiled as normal using `javac` and then the RI byte code enhancement tool would be invoked from the command line as follows:

```
javac Author.java
java com.sun.jdori.enhancer.Main Author.class Author.jdo
```

By default, the RI enhancement tool updates the Author byte code in place, and the original `Author.class` file is replaced with an enhanced one.

The initial JDO 1.0 specification didn't actually mandate the location and name of the XML metadata files. To aid in portability, the 1.0.1 maintenance revision to the specification changed this to what has just been described.

Prior to this revision, the name of a metadata file that contained multiple classes was the package name itself with a `.jdo` suffix, and it was located in the directory containing the package directory. In the Author example above, it would have been called `model.jdo` and would have been located in the `com/corejdo/examples/` directory. Some JDO implementations may still use this naming convention.

3.3.2 Mapping a class to the datastore

JDO does not define how to specify how the fields of a class should be mapped to the underlying datastore—for example, if using a relational database, what tables and columns should be used—nor does it define how the datastore schema should be created in the first place.

All this is datastore and JDO-implementation specific. A JDO implementation provides the necessary tools to allow a developer to define a mapping between

the persistence-capable classes and the underlying datastore and to define the required datastore schema.

As an example, a JDO implementation on top of a relational database might use the "vendor-extensions" element in the JDO metadata for a class to specify table and column names or define how field values should be stored. An implementation using an object database or a flat file might require only the Java classes and nothing more.

3.4 Connecting to a Datastore

After a persistence-capable class has been defined, the next step is to implement an application that actually uses it. Any JDO application must get a connection to the underlying datastore before it can do anything. This is done by configuring an instance of `PersistenceManagerFactory` and then using it to get a PersistenceManager.

`PersistenceManagerFactory` is just a Java interface; JDO does not specify how an actual `PersistenceManagerFactory` instance should be created. However, it does provide a helper class called JDOHelper that, among other things, provides a common bootstrap mechanism to create a PersistenceManagerFactory instance based on a specified set of properties:

```
static public PersistenceManagerFactory
  getPersistenceManagerFactory(Properties props)
```

The `getPersistenceManagerFactory()` method takes a set of properties as input and constructs a concrete instance of a `PersistenceManagerFactory`. The one required property is `javax.jdo.PersistenceManagerFactoryClass`, which is used to identify the name of an implementation's `PersistenceManagerFactory` class. All other properties are simply passed to the implementation's class and are implementation specific. These include things like connection URL, name, and password.

At a minimum, every JDO implementation must support bootstrapping via JDOHelper. Additionally, an implementation might provide its own PersistenceManagerFactory constructors that can be used to directly construct a PersistenceManagerFactory instance. It may also provide `PersistentManager` constructors to avoid having to go through a `PersistenceManagerFactory` at all. From a portability standpoint, these approaches are all JDO implementation specific.

`PersistenceManagerFactory` also implements `Serializable`. This allows a previously configured `PersistenceManagerFactory` instance to be located via JNDI.

After a PersistenceManagerFactory instance has been created, the `getPersitenceManager()` method can be used to get a `PersistenceManager` instance:

```
public PersistenceManager getPersistenceManager()
```

When an application is finished with a `PersistenceManager` instance, it should close it by calling the `close()` method:

```
public void close()
```

Connection Pooling

JDO leaves it up to the JDO implementation as to whether its **`PersistenceManagerFactory`** implements connection pooling. If implemented, the **`close()`** method on **`PersistenceManager`** may not physically close anything, but may instead return the PersistenceManager instance to the **`PersistenceManagerFactory`** to be reused.

The following code snippet taken from `MakeConnectionExample.java` shows how to get a `PersistenceManager` instance from a `PersistenceManagerFactory` and then close it:

```
import java.util.Properties;
import javax.jdo.*;

public class MakeConnectionExample {

  public static void main(String[] args) {

    Properties properties = new Properties();

    properties.put(
      "javax.jdo.PersistenceManagerFactoryClass", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionURL", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionUserName", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionPassword", "XXX");

    PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);

    PersistenceManager pm = pmf.getPersistenceManager();
```

```
    /* Do something interesting */

    pm.close();
  }
}
```

Of course, before this example can be used, the "XXX" strings would need to be replaced with values appropriate to the JDO implementation being used.

As an alternative to hard coding the properties as in the previous example, it is generally better to specify them as system properties or read them from a file. The Examples directory on the CD for this chapter contains examples of how these two alternative approaches would work. See the following directories for more information:

- MakeConnectionFromSystemPropertiesExample.java
- MakeConnectionFromFileExample.java

For a full list of the standard JDO property names that can be specified, see Chapter 5.

3.5 Creating an Object

After a `PersistenceManager` has been retrieved, it is possible to actually begin a transaction and do something interesting with the datastore. To begin a transaction, get the current `Transaction` instance from the `PersistenceManager`:

```
public Transaction currentTransaction()
```

And call the `begin()` method on `Transaction` to start a datastore transaction:

```
public void begin()
```

Typically, any interaction with a datastore is done within the context of a transaction. A transaction simply marks the beginning and end of a unit of work, whereby all changes made either happen or do not happen.

A transaction is also the mechanism that a datastore uses to enforce concurrency control between multiple applications accessing the datastore at the same time. Not all datastores support concurrent access, but those that do ensure that each ongoing transaction is isolated from the others. For more details on concurrency control and transactions, see Section 3.14.

Instances of persistence-capable classes can now be created and made persistent. They are constructed in the same manner as normal Java and, once constructed, need to be passed to the `makePersistent()` method on `PersistenceManager` to notify the JDO implementation that the instances need to be persisted:

```
public void makePersistent(Object pc)
```

The instances won't actually be stored in the datastore until the transaction is committed, which is done using the `commit()` method on `Transaction`:

```
public void commit()
```

Alternatively, a transaction can be rolled back—in this case, the instances won't become persistent and won't be stored in the datastore. This can be done using the `rollback()` method on `Transaction`, instead of `commit()`:

```
public void rollback()
```

The following code snippet taken from `CreateExample.java` shows how to create an instance of the Author class and make it persistent in the datastore:

```
Transaction tx = pm.currentTransaction();

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();
```

When they are successfully committed, the fields of the new Author instance are stored in the datastore, and the in-memory instance becomes what is termed "hollow." This means that its fields are cleared so that when the instance is used in the next transaction, they will be retrieved again from the datastore.

Object Lifecycle

JDO defines a number of lifecycle states for persistent objects ("hollow" being one of them). These states are used by the JDO implementation to manage the in-memory persistent objects. While it is not necessary to really understand the JDO object lifecycle (its of primary concern only to the JDO implementation), an appreciation of what happens under the covers is always a good idea. See Section 3.13 for more details.

Not every instance that needs to be persistent has to be passed to `makePersistent()`. JDO defines that any instance of a persistence-capable class that is reachable from a persistent object automatically becomes persistent also.

This means that only the root of a graph of objects actually needs to be passed to `makePersistent()`; all other objects in the graph implicitly become persistent also. This makes programming much easier.

3.6 Reading an Object

After an instance has been made persistent and its fields stored in the datastore, it can be retrieved again, either within the same application or by a different application.

There are three primary ways of finding an instance in the datastore with JDO—by navigation, via an Extent, or by a Query.

3.6.1 Reading by navigation

Retrieving an instance by navigation is simple. Extending the previous example, this code snippet taken from `ReadByNavigationExample.java` shows how the Author instance can be retrieved again in the next transaction simply by using it:

```
tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

Underneath the covers, the JDO implementation retrieves the fields for the Author instance from the datastore. The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

Using navigation to retrieve persistent objects from the datastore also applies when one persistent object is referenced from another. Essentially, anytime a persistent object is referenced, the JDO implementation retrieves the instance's fields from the datastore, if it has not already done so within the current transaction.

A common question often arises from the previous example: Why do the fields of the Author instance have to be retrieved from the datastore again after it was initially created?

The answer has to do with transactions. After each transaction ends (either by a commit or rollback), all in-memory persistent objects become hollow. This is because the representation of a persistent object in the datastore could be updated by another application after the transaction ends, making any in-memory values out of sync. For this reason, JDO's default behavior is to clear the fields of each persistent object and retrieve them again if used within the next transaction, thereby ensuring that the in-memory instance remains consistent with the datastore.

JDO has some advanced options that change this default behavior and allow instances to be retained over transaction boundaries or even accessed outside of a transaction altogether. See Chapter 5 for more details.

3.6.2 Reading using an extent

What happens if another application wants to access the new Author instance? Obviously, this application won't have a reference to which it can navigate. In this case, the application can use an Extent to find all Author instances.

Extents

An Extent represents a collection of all instances in the datastore of a particular persistence-capable class and can consist of just instances of the class or instances of all subclasses. An application can get an Extent for a class from a PersistenceManager and then use an Iterator to iterate through all instances, or it can use the Extent as input to a query.

By default, it is possible to get an Extent for any persistence-capable class. If an Extent is not required, then as a potential optimization, it is possible to specify explicitly that an extent is not required in the JDO metadata for the class. Depending on the underlying datastore, this may or may not make any difference.

An application can get an Extent using the `getExtent()` method on PersistenceManager:

```
public Extent getExtent(Class pc, boolean subclasses)
```

The subclasses flag determines whether the returned Extent represents instances of the specified class only or includes instances of subclasses as well. If true, then it includes instances of all subclasses as well.

The `iterator()` method on Extent can be used to get an Iterator that iterates through all the instances in the Extent:

```
public Iterator iterator()
```

The following code snippet taken from `ReadByExtentExample.java` shows how an Extent can be used to iterate through the instances of the Author class:

```
tx.begin();

Extent extent = pm.getExtent(Author.class, true);

Iterator authors = extent.iterator();

while (authors.hasNext()) {

  Author author = (Author) authors.next();

  String name = author.getName();
```

```
  System.out.println("Author's name is '" + name + "'.");
}

extent.close(authors);

tx.commit();
```

It is a good practice to use the `close()` or `closeAll()` method on Extent when an iterator has been used. The `close()` method closes an individual iterator, whereas `closeAll()` closes all iterators retrieved from the Extent:

```
public void close(Iterator iterator)
public void closeAll()
```

Depending on the underlying datastore, this may free resources that were allocated when the Iterator was created (a cursor, for example).

3.6.3 Reading using a query

An Extent is useful if an application wants to retrieve all instances of a class in no particular order. However, if an application is looking for a particular instance that has certain field values, an Extent is not very useful. In this case, a query is needed to find an instance based on the values of its fields.

JDO defines a query language called JDOQL (JDO Query Language). JDOQL is essentially a simplified version of the Java syntax for an "if" statement. The `PersistenceManager` is used to create a Query instance, and once created, a Query can be executed to return a collection of matching persistent objects. There are approximately nine methods on `PersistenceManager` to create a Query instance; the most straightforward requires only a Java class and a filter:

```
public Query newQuery(Class cln, String filter)
```

The class specifies the persistence-capable class that the query is for, and the filter, which is just a string, specifies which instances of the class should be returned.

A Query can also be created using an Extent. A query created using a Java class is equivalent to a Query created using an Extent whose subclasses property is true. This means that a Query created from a Java class may include instances of the specified class and subclass (which is generally the desired behavior).

If an application specifically wants to restrict the query to instances of a given class only, then it can create an Extent whose subclasses property is false and use the Extent to create the Query. When executed, the Query includes only instances of the class itself and ignores subclasses.

The basic syntax for a filter is as follows:

```
<field> <operator> <constant>
```

where <field> is the name of the field as declared in the Java class, <operator> is one of the standard Java comparison operators (==, !=, <, >, and so on), and <constant> is the value to be compared against. Much more sophisticated queries than this can be defined; see Chapter 6 for details.

The following code snippet taken from `ReadByQueryExample.java` shows how to use a query to find a previously created Author by name. For simplicity, it assumes that at least one author with the given name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

The `execute()` method on Query is used to execute the query and return the result. A half dozen ways of executing a query exist. The simplest is this:

```
public Object execute()
```

The `execute()` methods all return `java.lang.Object`, which must be cast to a java.util.Collection type for JDOQL queries. This collection is the set of instances that matched the specified filter. It can be iterated through in the normal manner to retrieve the persistent objects themselves.

The `execute()` methods on Query return `java.lang.Object` rather than `java.util.Collection` directly to allow for implementation-specific extensions.

JDO requires that JDOQL be supported; however, an implementation can optionally support alternative query languages (maybe SQL, OQL, or something proprietary). In this case, the result of executing a query may not be a

java.util.Collection; instead, it needs to be cast to something specific to the query language being used.

It is a good practice to use the close() method on Query to release the result returned from a call to execute():

```
public Object close(Object result)
```

Depending on the underlying datastore, this may free resources that were allocated during the execution of the query (a database cursor, for example).

3.7 Updating an Object

It is possible to modify a persistent object in the same manner as normal Java with no additional coding required. Assuming that the class defines methods that can be used to set the fields, then the application simply needs to invoke these methods within a transaction. The JDO implementation automatically detects when a field of a persistent object has been modified and marks the field as "dirty." At commit time, the dirty fields of all the modified persistent objects are written back to the datastore as part of the transaction.

The following code snippet taken from UpdateExample.java shows how to find a previously created Author instance and change its name. To validate that the name has changed in the datastore, it is printed in a new transaction. For simplicity, it assumes that at least one author with the specified name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close();

author.setName("Sameer Tyagi");

tx.commit();

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Sameer Tyagi'.
```

If it is decided that the changes should not be written to the datastore and should be discarded instead, then the `rollback()` method on Transaction can be used.

The following code snippet taken from `UpdateExampleWithRollback.java` is the same as the previous one, except that a rollback is used instead of a commit to undo the name change:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

author.setName("Sameer Tyagi");

tx.rollback(); // rollback() rather than commit()

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

Although the name of the Author instance is modified, because the transaction is rolled back, the modification is not persisted in the datastore. Therefore, when the instance is retrieved in the next transaction, the name is unchanged.

3.8 Deleting an Object

The last basic operation is deletion. Unlike Java, most datastores do not perform automatic garbage collection. Indeed, the semantics of what constitutes garbage when instances are made persistent changes. Just because a persistent object is no longer referenced by any other does not mean that it is garbage. Typically, it can always be retrieved at any time via Query or Extent.

JDO provides a mechanism to explicitly delete persistent objects. The `deletePersistent()` method on `PersistenceManager` can be used to permanently remove persistent objects from the datastore. That code looks like this:

```
public void deletePersistent(Object pc)
```

Of course, the persistent object isn't actually deleted until the transaction is committed. Rather, the JDO implementation marks the persistent object as deleted and, upon commit, requests that the datastore remove all the deleted persistent objects. The following code snippet taken from `DeleteExample.java` shows how to delete an instance of Author. For simplicity, it assumes that at least one author instance with the specified name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

pm.deletePersistent(author);

tx.commit();
```

Trying to access the fields of a deleted instance within the same transaction results in `JDOUserException` being thrown.

After the transaction commits, the in-memory instance reverts to being a normal transient Java object, with its fields reset to their default values. If a rollback is called instead, then the deletion is undone and the in-memory instance reverts to being a "hollow" persistent object again. The following code snippet taken from `DeleteWithRollbackExample.java` uses a rollback rather than a commit and then validates that the persistent object still exists by printing its name:

```
pm.deletePersistent(author);

tx.rollback(); // rollback() rather than commit()

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");
```

```
tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

JDO is unlike JVM, where garbage collection automatically determines whether an instance is still referenced and, if not, releases it. In JDO, however, it is up to the application to ensure that unwanted persistent objects are deleted from the datastore and that a persistent object being deleted is no longer referenced by another.

Depending on the datastore, it may be possible to define constraints to guard against deleting instances that are still referenced, but this is implementation specific. If a persistent object does reference a previously deleted persistent object, then `JDOUserException` is thrown if the deleted persistent object is later accessed (because it no longer will be found in the datastore).

There is no equivalent of "persistence by reachability" when deleting. Deleting a persistent object deletes only the specified instance; it does not automatically delete any referenced instances.

It is possible for an application to implement a cascading delete behavior by using the JDO `InstanceCallbacks` interface and implementing the `jdoPreDelete()` method to explicitly delete referenced instances. See Chapter 5 for more details.

3.9 JDO Object Model

So far, the examples have used a very simple Java class to demonstrate basic concepts. JDO can, of course, be used with much more sophisticated classes than this. In addition to supporting fields of primitive Java types and system classes like String, JDO can handle object references, Java collections, and inheritance of both classes and interfaces.

This section outlines what can and can't be used when developing a persistence-capable class, including the following:

- The basic field types that can be used within a persistence-capable class
- Using references to persistence-capable classes, non-persistence-capable classes, and interfaces
- Using standard Java collections
- Using arrays
- Support for inheritance

- The class and field modifiers that can be used
- What JDO doesn't support

3.9.1 Basic types

A persistence-capable class can have fields whose types can be any primitive Java type, primitive wrapper type, or other supported system immutable and mutable classes. Table 3-1 provides a summary of these basic types.

Table 3–1 Supported Basic Types

Primitive Types	Wrapper Classes	Supported System Interfaces	Supported System Classes
boolean	java.lang.Boolean¥	java.util.Collection§	java.lang.String¥
byte	java.lang.Byte¥	java.util.List*§	java.math.BigDecimal¥
char	java.lang.Character¥	java.util.Map*§	java.math.BigInteger¥
double	java.lang.Double¥	java.util.Set§	java.util.Date
int	java.lang.Integer¥	javax.jdo.PersistenceCapable	java.util.Locale¥
float	java.lang.Float¥		java.util.ArrayList*§
long	java.lang.Long¥		java.util.HashMap*§
short	java.lang.Short¥		java.util.HashSet§
			java.util.Hashtable*§
			java.util.LinkedList*§
			java.util.TreeMap*§
			java.util.TreeSet*§
			java.util.Vector*§

*Support is an optional JDO feature.
§See Section 2.9.3 for more details on using collection classes.
¥Immutable class (Its value can't be changed.)

If a persistence-capable class has fields of any of the above types, then by default these fields are persisted in the datastore (providing they are not declared as transient). If a field is a reference to a persistence-capable class, then by default these fields are persisted also. Fields of any other types are not persisted in the datastore by default and need explicit metadata to indicate that they should be persisted. The types of these fields might be a reference to the Java interface, a `java.lang.Object`, a non-supported system class/interface, or a non-persistence-capable class.

JDO defines both mandatory and optional features. A JDO implementation is said to be compliant if it supports all the mandatory features defined by the JDO specification. An implementation may support one or more optional features, in which case it must adhere to what is defined for the optional feature by the JDO specification if it is to be JDO compliant.

From a portability standpoint, an application should not rely on support of an optional feature. But should a particular feature be required, then at least the application is still portable across implementations that support the feature or set of features used.

See Chapter 5 for more details on specific optional features.

3.9.2 References

A persistence-capable class can have fields that reference instances of other persistence-capable classes. The following code snippet shows a Book class that has an "author" field, which is a reference to an instance of Author:

```
public class Book {

  private String name;
  private Author author;

  public Book(String name, Author author) {

    this.name = name;
    this.author = author;
  }

  protected Book() {}

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public Author getAuthor() {

    return author;
  }

  public void setAuthor(Author author) {
```

```
    this.author = author;
  }
}
```

The revised JDO metadata for this class in `Book.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Book"/>
  </package>
</jdo>
```

The following code snippet taken from `CreateWithReachabilityExample.java` shows persistence by reachability at work. Both an Author and a Book instance are created. The Book instance is added to the Author instance, and the Author instance is explicitly made persistent, so the Book instance is automatically made persistent because it is reachable from the Author instance:

```
tx.begin();

Author author = new Author("Keiron McCammon");

Book book =
  new Book("Core Java Data Objects", "0-13-140731-7");

author.addBook(book);

pm.makePersistent(author);

tx.commit();
```

As well as supporting references to persistence-capable classes, JDO also supports references to Java interfaces and even a `java.lang.Object`. Because fields of these types are not persistent by default, additional metadata is required to denote that the field should be persistent. Any referenced instances should still be instances of a persistence-capable class. (Support for references to non-persistence-capable classes is a JDO optional feature.)

The following code snippet shows a revised Book class that has an “author” field, which now is of type `java.lang.Object` instead of Author:

```
public class Book {

  private String name;
```

```
  private Object author; // Uses Object rather than Author

  /* Rest of code not shown */
}
```

The major difference here compared to the previous example is the metadata in Book.jdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Book">
      <field
          name="author"
          persistence-modifier="persistent"/>
    </class>
  </package>
</jdo>
```

Because the field's type is java.lang.Object, it is not persistent by default. It must be explicitly denoted as persistent in the metadata for the class. The same would be true if it were a reference to an interface.

A JDO implementation may optionally support references to instances that do not belong to a persistence-capable class. If not supported, a Java class cast exception would be thrown at the time of assignment. (An application intended to run against multiple JDO implementations should not rely on support for persisting non-persistence-capable classes.)

If references to non-persistence-capable classes are supported, then these instances exist in the datastore only as part of the referencing persistent object. It is the responsibility of the application to inform the JDO implementation when an instance of a non-persistence-capable class is modified because the JDO implementation is unable to automatically detect when the field of a non-persistence-capable class is changed.

The JDO implementation can't detect when an instance of a non-persistence-capable class is modified because the class doesn't adhere to the persistence-capable programming style. A persistence-capable class automatically informs the JDO implementation before a field is changed (this is what the byte code enhancement process takes care of), but a non-persistence-capable class does not.

If the JDO implementation is not explicitly notified of changes to non-persistence-capable classes, then the modifications do not get written to the datastore on commit. The easiest way to notify a JDO implementation that a non-persistence-capa-

ble instance has been modified is to use the `makeDirty()` method on JDOHelper:

```
static void makeDirty(Object pc, String field)
```

The first argument is the persistent object that references the non-persistence-capable instance. The second argument is the field name of the reference.

JDOHelper

JDOHelper was previously introduced as a way to bootstrap a JDO application and get an instance of a PersistenceManagerFactory. As well as this, JDOHelper has a number of additional methods that act as shortcuts to the various JDO APIs.

The following code snippet shows a non-persistence-capable class called Address that can be used to store the address of an Author:

```
import java.io.Serializable;

public class Address implements Serializable {

  private String street;
  private String zip;
  private String state;

  public Address(String street, String zip, String state) {

    this.street = street;
    this.zip    = zip;
    this.state  = state;
  }

  /* Additional getters and setters not shown */

  public void setZip ( String zip ) {

    this.zip = zip;
  }
}
```

Exactly how a JDO implementation manages the fields of non-persistence-capable classes is undefined by JDO. Some implementations may require that a class implement Serializable or follow the JavaBean pattern, or may require that its fields be declared public.

The following code snippet shows a revised Author class that contains a reference to an Address and shows how to set the zip code of the address. The method first calls `makeDirty()` to inform the JDO implementation that the address is being be modified:

```
import javax.jdo.*;

public class Author {

  private String  name;
  private Address address;

  public Author(String name, Address address) {

    this.name    = name;
    this.address = address;
  }

  protected Author () {}

  public void setZip(String zip) {

    JDOHelper.makeDirty(this, "address");

    address.setZip(zip);
  }
}
```

JDOHelper.makeDirty()

It is a best practice to call **`makeDirty()`** before making any modifications to a non-persistence-capable instance. Depending on the underlying datastore, the call to **`makeDirty()`** may result in a concurrency conflict indicating that changes aren't allowed at that time.

The revised JDO metadata for this class in `Author.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author">
      <field
          name="address"
          persistence-modifier="persistent"/>
    </class>
```

```
  </package>
</jdo>
```

Because the address field is a reference to a non-persistence-capable class, it is necessary to specifically specify that the field should be persistent.

When using non-persistence-capable instances, it is best to modify them only through methods on the referencing persistent object. This way, the persistent object can call `makeDirty()` on itself before making any changes. If a non-persistence-capable object is passed by reference and modified elsewhere, it becomes hard for the application to ensure that the referencing persistent object is notified of the changes, in which case any changes are ignored.

3.9.3 Collection classes

A persistence-capable class can have fields that reference the standard Java collection classes. Instances of these collection classes exist in the datastore only as part of the referencing persistent object. At a minimum, JDO mandates support for `java.util.HashSet`; support for `ArrayList`, `HashMap`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, and `Vector` is optional. However, most JDO implementations support all these collection classes.

Optional Features

To determine whether an implementation supports the optional collection classes, use the **`supportedOptions()`** method on PersistenceManagerFactory. See Chapter 5 for more details.

The following code snippet shows a revised Author class that uses a `java.util.HashSet` to hold the books that an author has written:

```
import java.util.*;

public class Author {

  private String name;
  private Set books = new HashSet();

  public Author (String name) {

    this.name = name;
  }

  protected Author () {}
```

```
  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public void addBook(Book book) {

    books.add(book);
  }

  public Iterator getBooks() {

    return books.iterator();
  }
}
```

The revised JDO metadata for this class in `AuthorWithBooks.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author">
      <field name="books">
        <collection
            element-type=
              "com.corejdo.examples.model.Book"/>
      </field>
    </class>
  </package>
</jdo>
```

Because this class uses a Java collection, additional metadata can be used to declare the class of the instances that is stored in the collection. Because Java collections are un-typed, it is impossible for a JDO implementation to determine this information from the Java class itself.

Additional Metadata

The additional metadata shown for the "books" field is optional; it does not need to be specified. If not specified, the JDO runtime assumes that the collection may contain a reference to any persistent object, as does normal Java.

Even though it is optional, it is best practice to define the element type of a collection because it potentially allows the JDO runtime to optimize how it handles the field both in-memory as well as in the datastore.

Unlike user-defined, non-persistence-capable classes, JDO mandates that the supported collections automatically detect changes and notify the referencing persistent object. A JDO implementation does this by replacing instances of system-defined collection classes with instances of its own equivalent classes that perform this detection. For a new persistent object, this replacement happens during the call to `makePersistent()`; for existing persistent objects, this occurs when they are retrieved from the datastore. All this is completely transparent to the application. It just means that an application should not reply on a collection being a particular concrete Java class (`java.util.HashMap`, for example).

3.9.4 Arrays

Support for a persistence-capable class with fields that use Java arrays is optional in JDO. If supported, arrays are similar in nature to user-defined, non-persistence-capable classes except that a JDO runtime may automatically detect changes and notify the referencing persistent object of the modification. For portability, an application should assume responsibility for notifying the referencing persistent object when an array is modified.

With non-persistence-capable classes, it is a best practice to modify arrays only through methods on the referencing persistent object. This way, the referencing persistent object can call `makeDirty()` on itself before making any change. If an array is passed by reference and modified elsewhere, it becomes hard for the application to ensure that the owning persistent object is notified of the changes, in which case any changes are ignored.

The following code snippet shows a revised Author class that uses an array to store the books that an author has written. As a simplification, the array holds only a single book; in real life, any sized array could have been used:

```
import java.util.*;

public class Author {
```

```
  private String name;
  private Book   books[] = new Book[1];

  public Author(String name) {

    this.name = name;
  }

  protected Author()

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public void addBook(Book book, int i) {

    JDOHelper.makeDirty(this, "books");

    books[i] = book;
  }

  public Book getBook(int i) {

    return books[i];
  }
}
```

The revised JDO metadata for this class in `Author.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"/>
  </package>
</jdo>
```

Because the field is an array of a persistence-capable class, no additional metadata is required and the array field is persistent by default.

3.9.5 Inheritance

JDO supports inheritance of both interfaces and classes. Because Java interfaces don't define any fields, a JDO implementation needs to do nothing, so Java interfaces can be used with no real consideration as far as JDO is concerned.

When using class inheritance (abstract or otherwise), JDO needs to be aware of the implementation hierarchy, because each class can define its own fields. In the usual case, where all classes in the inheritance hierarchy are themselves persistence-capable, it is straightforward: The only requirement is to denote the persistence-capable superclass of a class in the JDO metadata.

The following code snippet shows a class that extends the Book class:

```
public class RevisedBook extends Book {

  private Book original;

  public RevisedBook(String name, Book original) {

    super(name);

    this.original = original;
  }

  protected RevisedBook() {}

  public Book getOriginal() {

    return original;
  }

  public void setOriginal(Book original) {

    this.original = original;
  }
}
```

The JDO metadata for this class in `RevisedBook.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class
        name="RevisedBook"
             persistence-capable-superclass="Book"/>
  </package>
</jdo>
```

The additional metadata denotes the persistence-capable superclass of the class. The persistence-capable-superclass name uses the Java naming rules: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class.

It is possible for a persistence-capable class to extend a non-persistence-capable class. In this case, any fields defined by the non-persistence-capable class are ignored by JDO and are not persisted in the datastore. It is also possible to have a persistence-capable class extend a non-persistence-capable class, which itself extends a persistence-capable class. In all cases, any fields of a non-persistence-capable class are ignored by JDO; it is the responsibility of the application to manage these fields.

It is best to make all classes in an inheritance hierarchy persistence-capable. This avoids added complications. If this isn't possible and non-persistence-capable classes have to be used, then a persistence-capable subclass should implement the `InstanceCallbacks` Interface and use the `jdoPostLoad()`, `jdoPreStore()`, and `jdoPreClear()` methods to manage the fields of the non-persistence-capable classes explicitly.

3.9.6 Modifiers

JDO supports all Java class and field modifiers—private, public, protected, static, transient, abstract, final, synchronized, and volatile—with the following caveats:

- A field declared as transient is not persisted it in the datastore by default. This default behavior can be overridden by explicitly declaring the field as persistent in the class's metadata.
- A field declared as final cannot be persisted in the datastore; its value can be set only during construction.
- A field declared as static cannot be persisted in the datastore; it has meaning only at the class level within the JVM.

Apart from static and transient, use of field modifiers is essentially orthogonal to the use of JDO.

3.9.7 What isn't supported

JDO supports most of what can be defined in Java. The major exception is that JDO can't make Java system classes persistence-capable; rather, it mandates support for specific system classes (as already outlined) as fields of a persistence-capable class. Also, classes that depend on an inaccessible or remote state, such as those that use JNI or those that extend system-defined classes, cannot be made persistence-capable and cannot be used as fields of persistence-capable classes.

3.10 Exception Handling

All the examples so far have ignored the possibility of exceptions. JDO defines a set of exceptions that can be thrown by the JDO implementation. All JDO exceptions are declared as runtime exceptions because they can be thrown at anytime, not just when calling JDO methods. As an example, navigating from one instance to another may result in an exception if there is a problem with accessing the datastore when retrieving the fields of an instance.

JDO classifies exceptions broadly into fatal or non-fatal exceptions. Non-fatal exceptions indicate that an operation failed but can be retried, whereas fatal exceptions indicate that the only recourse is to start again. Beyond this, there are user exceptions (fatal or non-fatal), which indicate user error; datastore exceptions (fatal or non-fatal), which indicate a datastore error; an internal exception (fatal), which indicates a problem with the JDO implementation; and an unsupported feature exception (non-fatal), which indicates that the JDO implementation doesn't support a particular feature.

An application should ensure that it handles all exceptions correctly, not just JDO exceptions. In particular, if connection pooling is being used, an application should ensure that a PersistenceManager instance is closed properly even if an exception is thrown. The following code snippet taken from `CreateWithExceptionsExample.java` shows how to use a try/final block to catch any exception and ensure that the `PersistenceManager` instance is closed:

```
PersistenceManager pm = null;

try {

  pm = pmf.getPersistenceManager();

  Transaction tx = pm.currentTransaction();

  tx.begin();

  Author author = new Author("Keiron McCammon");

  pm.makePersistent(author);

  tx.commit();

  pm.close();
}

finally {
```

```
  if (pm != null && !pm.isClosed()) {

    if (pm.currentTransaction().isActive()) {

      pm.currentTransaction().rollback();
    }

    pm.close();
  }
}
```

In this simple example, it really doesn't matter whether the transaction is rolled back or the `PersistenceManager` is closed because it exits immediately thereafter anyway.

3.11 Object Identity

One of the key concepts defined in JDO is that of object identity, although in most cases a developer is not even aware that it exists. Every persistent object has a JDO object identity. This identity associates the in-memory Java object with its representation in the underlying datastore. JDO ensures that there is only one in-memory representation of a given persistent object for a given PersistenceManager. This is known as "uniquing." Uniquing ensures that no matter how many times a persistent object is found, it has only one in-memory representation. All references to the same persistent object within the scope of the same PersistenceManager instance reference the same in-memory object.

The following code snippet taken from `UniquingExample.java` shows uniquing at work. It creates a new Author instance, begins a new transaction, and finds the Author again using a query. The two references are then compared to validate that they both refer to the same in-memory object:

```
tx.begin();

Author author1 = new Author("Keiron McCammon");

pm.makePersistent(author1);

tx.commit();

tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();
```

```
Author author2 = (Author) result.iterator().next();

tx.commit();

if (author1 == author2)
  System.out.println("There is only one object in memory");
```

The output would be as follows:

```
There is only one object in memory
```

However, because it is possible to create multiple PersistenceManager instances within a JVM, it is possible that a persistent object may have multiple in-memory representations at any given time—at most, one per PersistenceManager. Each would have the same JDO object identity, but would be a different in-memory Java object. To determine whether two in-memory objects represent the same persistent object, their JDO object identities can be compared. The JDOHelper class provides a method to get the JDO object identity of an object:

```
static Object getObjectId(Object pc)
```

The returned object can be compared with another using the `equals()` method to determine whether two in-memory objects represent the same persistent object in the datastore.

The following code snippet taken from `ObjectIdentityExample.java` creates an Author using one `PersistenceManager`, and then using a different `PersistenceManager`, it finds the Author again. The two references are compared to validate that they refer to different in-memory objects. The JDO identities are then compared to validate that they do, however, represent the same persistent object:

```
tx1.begin();

Author author1 = new Author("Keiron McCammon");

pm1.makePersistent(author1);

tx1.commit();

PersistenceManager pm2 = pmf.getPersistenceManager();

Transaction tx2 = pm2.currentTransaction();

tx2.begin();

Query query =
```

```
  pm2.newQuery(Author.class, "name==\"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author2 = (Author) result.iterator().next();

tx2.commit();

if (author1 != author2)
  System.out.println(
    "There are multiple objects in memory");

Object emp1Id = JDOHelper.getObjectId(author1);
Object emp2Id = JDOHelper.getObjectId(author2);

if (emp1Id.equals(emp2Id))
  System.out.println("But they represent the same Author");
```

The output would be as follows:

```
There are multiple objects in memory
But they represent the same Author
```

JDO actually defines three types of object identity for persistent objects: datastore identity, application identity, and non-durable identity.

3.12 Types of Object Identity

The JDO specification requires that either datastore or application identity be supported (support for both is optional) and support for non-durable identity is optional. Most JDO implementations support datastore identity as their default, and some also support application identity (notably, those that are designed to run on top of a relational database).

3.12.1 Datastore identity

Datastore identity is the simplest form of identity. The JDO object identity of a persistent object is assigned and managed by the JDO implementation, typically in conjunction with the underlying datastore. The identity is orthogonal to the values of any of the fields of a persistent object.

There is nothing an application itself needs to do or provide when using datastore identity other than the Java classes themselves.

Unless there is a specific need otherwise, datastore identity should always be used. It's the simplest and most portable form of JDO object identity, and it is widely supported by most JDO implementations.

3.12.2 Application identity

Application identity is more akin to the concept of a relational primary key. The JDO object identity of a persistent object is determined by the values of certain of its fields.

If using application identity, the application needs to specify which fields are part of the primary key in the JDO metadata for the class and provide a class that can be used by a JDO implementation to represent the persistence-capable class's object identity.

Application identity is particularly useful if using JDO on top of a predefined datastore schema where primary keys have already been defined.

The following code snippet shows how the Author class does not need to change from the original Author class. Regardless of the type of identity being used, the persistence-capable class remains the same:

```
public class Author {

  private String name;

  public Author (String name) {

      this.name = name;
  }

  protected Author () {}

  public String getName () {

    return name;
  }

  public void setName (String name) {

    this.name = name;
  }
}
```

However, the JDO metadata for the class does change. The `identity-type` attribute needs to be set to "application" and the `objectid-class` attribute should denote the class that the JDO implementation can use to represent the object identity of a persistent object. Also, the `primary-key` attribute should be set to "true" for each field of the persistence-capable class that is part of the primary key.

The revised JDO metadata for the Author class in `Author.jdo` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"
        identity-type="application"
        objectid-class="AuthorUsingApplicationIdentityKey">
      <field
          name="name"
          primary-key="true"/>
    </class>
  </package>
</jdo>
```

If the package of the object identity class is not specified, it is assumed to be the same as the persistence-capable class itself.

The final requirement when using application identity is to provide the implementation of the object identity class. JDO mandates that this class adhere to the following rules:

- The class must be public.
- The class must implement `Serializable`.
- The class must have a public `no-arg` constructor.
- There must be a public field of the same name and type as each field that is part of the primary key for the persistence-capable class.
- The `equals()` and `hashCode()` methods must use the values of all the primary key fields.
- The `toString()` method must return a string representation of the object identity instance.
- The class must have a constructor that takes a string returned from the `toString()` method of an object identity instance and constructs an equivalent object identity instance.

The following code snippet taken from `AuthorKey.java` shows the implementation of the object identity class for the Author persistence-capable class:

```
import java.io.Serializable;

public class AuthorKey implements Serializable {

  /*
   * JDO requires that an identity class have a
   * public field of the same name and type for each of
```

```
   * the primary key fields of the persistence-capable
   * class.
   */
  public String name;

  /*
   * JDO requires a public no-arg constructor
   */
  public AuthorKey () {}

  /*
   * JDO requires a public constructor that takes a string.
   */
  public AuthorKey (String oid) {

    name = oid;
  }

  /*
   * JDO requires that the toString() method return
   * a string representation of the primary key fields that
   * can later be used to recreate an identity instance
   * using the string constructor.
   */
  public String toString() {

    return name;
  }

  /*
   * JDO requires the equals() method to compare based on
   * all the primary key fields.
   */
  public boolean equals(Object obj) {

    return name.equals(
     ((AuthorUsingApplicationIdentityKey) obj).name);
  }

  /*
   * JDO requires the hashCode() method to return a
   * hashcode based on all the primary key fields.
   */
  public int hashCode() {

    return name.hashCode();
  }
}
```

Primary Key Fields

A persistence-capable class can have multiple primary-key fields. The **`equals()`** and **`hashCode()`** methods on the object identity class must take into account each of the primary-key fields. Also, the **`toString()`** method should concatenate all the primary-key fields into a single string, which the string constructor can later tokenize to recreate the identity instance again.

When using application identity, only fields declared in abstract superclasses and the least least-derived concrete class in the inheritance hierarchy can be part of the primary key. For example, if a class inherited from the Author class, then it could not provide its own object identity class or set the "primary-key" attribute to "true" for any of its locally declared fields. Instead, it would use the same application identity as defined for the Author class.

If using application identity, a JDO implementation may optionally allow an application to change a persistent object's identity by modifying the values of the primary key fields. If not supported, a `JDOUnsupportedOptionException` is thrown.

3.12.3 Non-durable identity

Non-durable identity is essentially no identity. The JDO object identity of a persistent object is valid only within the context of a given transaction. If the same persistent object is accessed in a different transaction, it may have a different object identity.

Non-durable identity should be used only when object identity has no relevance, i.e., when there is no need to maintain references between objects. Essentially, it only makes sense to query for objects with non-durable identity.

There are few, if any, JDO implementations that actually support non-durable identity. Its use is limited to a small number of use cases, and as such, most developers can safely ignore non-durable identity.

3.13 Object Lifecycles

JDO defines a number of different states in which an in-memory persistent object can be. Principally, these states are used by the JDO implementation to determine whether the fields of an instance need to be retrieved from the datastore and, if modified, whether they need to be written back. Together, these states represent an in-memory persistent object's lifecycle with respect to JDO.

Chapter 4 describes the object lifecycle states in greater detail; however, in general, only the following basic states need to be understood:

- Transient
- Persistent
- Hollow

Figure 3-4 shows a simplified state diagram for these basic lifecycle states:

Transient denotes a normal, non-persistent Java object. When an instance of a persistence-capable class is created, it starts life as being transient. By default, JDO does not do anything to manage transient instances.

When made **persistent,** either explicitly via `makePersistent()` or because an instance is reachable from another persistent object, an instance transitions to being persistent (it actually transitions to a "persistent-new" state). Five different states are used to represent persistence. These allow the JDO implementation to differentiate between whether an instance is new ("persistent-new"), has been modified ("persistent-dirty") or deleted ("persistent-deleted"), and so on. All that really matters is that the instance is persistent, it has a JDO object identity, and its fields have been retrieved from the datastore within the context of the current transaction.

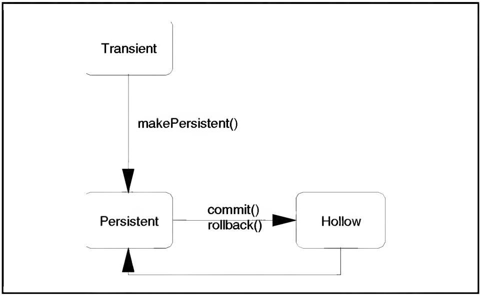

Figure 3–4 The simplified lifecycle state diagram.

A **hollow** instance is also a persistent object (it has a JDO object identity); however, its fields have not yet been retrieved from the datastore, hence the term "hollow." The first time that a hollow instance is accessed, its fields are retrieved from the datastore within the context of the current transaction and the instance transitions to being persistent. (It actually transitions to "persistent-clean" to indicate the instance hasn't yet been modified.) Likewise, by default, after a transaction ends, all instances that are persistent transition to being hollow.

There are several optional features that provide support for transient instances being managed as transactional objects (changes to their fields being undone on rollback) and persistent objects being accessed outside of transactions. See Chapter 5 for more details.

3.14 Concurrency Control

All the examples so far have been single-user applications (only one transaction was accessing the datastore at a time). It is, of course, possible for multiple applications to be running concurrently. The JDO specification does not make any specific statements about concurrency control, because it is a feature of the underlying datastore rather than of JDO itself.

JDO does define the concept of a *datastore transaction* and says that access to the datastore should be done within the context of a transaction and that these transactions should exhibit ACID properties. This allows the JDO implementation to use the underlying datastore's own concurrency control mechanisms, which may be based on a pessimistic or optimistic locking scheme or some other approach. Because of this, JDO does not expose APIs to explicitly lock persistent objects. If locks are required by the underlying datastore to enforce concurrency control, then it is the responsibility of the JDO implementation to manage them implicitly.

3.14.1 ACID transactions

ACID stands for Atomic, Consistent, Isolated, and Durable, and it defines standard properties for a transaction:

> **Atomic:** Upon commit, either all changes are in the datastore or none of them are in the datastore. This means that, should something fail, arbitrary changes aren't left behind in the datastore.
>
> **Consistent:** Prior to and after commit, all data is consistent as far as the datastore is concerned. This means that all data accessed during a transaction is initially consistent and that, after commit, all changes leave the data consistent. Consistent means that the data does not violate any datastore constraints or invariants.

Isolated: Prior to commit, any changes made are not visible to other transactions and vice versa. This means that one transaction is not able to see changes made by another transaction; it can see only committed changes. Many datastores support varying levels of isolation that trade off strict transaction isolation against improved concurrency.

Durable: After commit, any changes are guaranteed to persist even in the event of failure. This means that when a transaction ends, any changes are in the datastore.

Different transactional datastores use different mechanisms to enforce ACID transactions. Therefore, when using JDO, when and how concurrency conflicts are detected and resolved is dependent upon which underlying datastore is being used. All that can be guaranteed is that a transaction is ACID in nature.

3.14.2 Optimistic transactions

JDO provides optional support for optimistic transactions. Optimistic transactions are useful for applications that have long running transactions where it is undesirable to hold datastore resources (such as locks) for an extended period of time.

With optimistic transactions, all concurrency control is deferred until the end of the transaction. See Chapter 5 for more details on optimistic transactions.

3.15 Summary

After reading this chapter, you should understand what transparent persistence is all about and what it means when a Java class is persistence-capable. You have seen how to create, read, update, and delete persistent objects, and you have explored how JDO supports the full Java object model: basic types, references, collection classes, and inheritance. You should also understand some of the more advanced JDO concepts like handling exceptions, object identity, object lifecycles, and concurrency control.

In the next chapter, you learn more about the different states in the JDO object lifecycle and how the JDO implementation manages persistent object in-memory.